A Three-tier Strategy for Reasoning about Floating-Point Numbers in SMT

Sylvain Conchon, Mohamed Iguernelala, Kailiang Ji, Guillaume Melquiond, Clément Fumex

A Three-tier Strategy for Reasoning about Floating-Point Numbers in SMT*

Sylvain Conchon[2,3], Mohamed Iguernlala[1,2], Kailiang Ji[2], Guillaume Melquiond[3], and Clément Fumex[2,3]

[1] OCamlPro SAS, Gif-sur-Yvette F-91190
[2] LRI (CNRS & Univ Paris-Sud), Université Paris-Saclay, Orsay F-91405
[3] Inria, Université Paris-Saclay, Palaiseau F-91120

Abstract. The SMT-LIB standard defines a formal semantics for a theory of floating-point (FP) arithmetic (FPA). This formalization reduces FP operations to reals by means of a *rounding operator*, as done in the IEEE-754 standard. Closely following this description, we propose a three-tier strategy to reason about FPA in SMT solvers. The first layer is a purely axiomatic implementation of the automatable semantics of the SMT-LIB standard. It reasons with exceptional cases (*e.g.* overflows, division by zero, undefined operations) and reduces finite representable FP expressions to reals using the rounding operator. At the core of our strategy, a second layer handles a set of lemmas about the properties of rounding. For these lemmas to be used effectively, we extend the instantiation mechanism of SMT solvers to tightly cooperate with the third layer, the NRA engine of SMT solvers, which provides interval information. We implemented our strategy in the Alt-Ergo SMT solver and validated it on a set of benchmarks coming from the SMT-LIB competition, but also from the deductive verification of C and SPARK programs. The results show that our approach is promising and compete with existing techniques implemented in state-of-the-art SMT solvers.

Keywords: SMT, Floating-point arithmetic, Program verification

1 Introduction

Floating-point (FP) numbers is a common way of approximating reals in a computer. However, due to their finite and discrete representation, rounding errors are inherent to FP operations and the result of an FP computation may overflow or diverge from the exact value expected on reals. Because such deviations may cause serious software and hardware failures, it is important to develop tools that help analyze programs with FP numbers.

FP reasoning has been intensively investigated in the past in the context of theorem proving [7, 8, 13], abstract interpretation [12] or constraint solving [15].

* This work is supported by the ANR projects SOPRANO (ANR-14-CE28-0020) and ProofInUse (ANR-14-LAB3-0007).

Recently, several decision procedures have been proposed in the context of *Satisfiability Modulo Theories* (SMT). In fact, the SMT-LIB standard [2] includes a formal semantics for a theory of floating-point arithmetic (FPA) [5] and some SMT solvers already support it (*e.g.* MathSAT5, REALIZER, SONOLAR, Z3).

There are three kinds of techniques for integrating FP reasoning in SMT. The first one interprets FP numbers as bit-vectors and FP operations as Boolean circuits. The second technique consists in lifting the CDCL procedure at the heart of SMT to an abstract algorithm (ACDCL) manipulating FP numbers as abstract values. In [4], real intervals are used to overapproximate FP values. The Boolean Constraint Propagations (BCP) of CDCL is extended with Interval Constraint Propagation (ICP) and decision steps make a case analysis by interval splitting. The third technique follows the IEEE-754 standard which reduces FP numbers to reals by means of a rounding operator. In [14], a two-layer decision procedure is presented. The first layer replaces FP terms by equal-valued exact-arithmetic terms in a fragment of real/integer arithmetic (RIA) extended by ceiling and floor functions. The second layer performs rounding which requires a case analysis to determine the binades of real values.

All these techniques suffer from drawbacks. Consider for instance the following FP formula[4] on binary64 FP numbers, where $\oplus$ is the FP addition with default *rounding to nearest with tie breaking to even* mode.

$$\texttt{-2.} \preceq u \preceq \texttt{2.} \wedge \texttt{-2.} \preceq v \preceq \texttt{2.} \wedge \texttt{-1.} \preceq w \preceq \texttt{1.} \wedge u \preceq v \wedge v \oplus w \prec u \oplus w \quad (1)$$

With bit-blasting, Z3 takes 5 minutes to prove unsatisfiability of this conjunction and MathSAT5 times out after 10 minutes. MathSAT5 also times out when using ACDCL, and so does REALIZER when using RIA. Possible explanations for these results are the following. On large bit-width FP numbers, bit-blasting techniques tend to generate very large propositional formulas hardly tractable even by very efficient SAT solvers. Concerning the second approach, ACDCL efficiency strongly depends on the choice of the abstract domain: the less precise (or expressive) the domain is, the more case splits are needed. On this example, the tool is forced to enumerate aggressively to offset the weakness of the interval-based domain. About the third technique, only few SMT solvers support RIA (with ceiling and floor functions) and providing an efficient and complete decision procedure for it is still an active domain of research. Here, to conclude rapidly, the RIA engine should be powerful enough to prove that the integral rounding functions used in the translation are monotonic.

In this paper, we propose another approach which takes the form of a three-tier strategy. Similarly to the third technique, we closely follow the SMT-LIB standard by reducing FP operations to reals using an explicit rounding operator. But instead of encoding rounded terms in RIA, we keep the operator and reason about it modulo a set of lemmas, mainly borrowed from a FP library of theorem prover such as Flocq [3].

Our first layer translates FP expressions into expressions mixing real numbers (for operations on finite FP inputs) and Booleans (for encoding exceptional val-

[4] Inspired by a discussion on `https://github.com/Z3Prover/z3/issues/823`

ues). For the example of Formula (1), some propositional simplifications reduce the translated expressions into the following real-only formula:

$$-2 \leq \overline{u} \leq 2 \ \wedge \ -2 \leq \overline{v} \leq 2 \ \wedge \ -1 \leq \overline{w} \leq 1 \ \wedge \ \overline{u} \leq \overline{v} \ \wedge \ \circ(\overline{v} + \overline{w}) < \circ(\overline{u} + \overline{w})$$

where $\overline{x}$ denotes the real value of a finite FP number x and where $\circ(x)$ is the FP number (seen as a real) chosen to represent a real x for a given format according to the default rounding mode. The second layer contains a lemma about the monotonicity of $\circ$, that is: $\forall x, y : \mathbb{R}.\ x \leq y \implies \circ(x) \leq \circ(y)$. Using contrapositive of monotonicity, we deduce from the last literal that $\overline{v} + \overline{w} < \overline{u} + \overline{w}$, which contradicts $\overline{u} \leq \overline{v}$, thanks to a simple reasoning over real numbers conducted by the third layer.

While our strategy only requires NRA support, the main challenge is to find the good instances of lemmas about rounding. Unfortunately, the syntactic (modulo uninterpreted equalities) matching algorithm of SMT is too weak to find relevant instances for some of them. Indeed, problematic lemmas require interval information to be instantiated efficiently. We expect the NRA decision procedure to abstract FP terms by real intervals, as in the ACDCL approach, and to export those intervals to the matching algorithm.

Since our appoach is based on a generic instantiation mechanism, it is not meant to be complete. In practice, this is not a limitation, as we are mainly interested in discharging verification conditions coming from program verification, that is, in proving the unsatisfiability of formulas.

Contributions: We make the following contributions.

1. We propose a three-layer architecture for extending SMT solvers with FP reasoning. The heart of our approach is a new mechanism for reasoning about quantifiers modulo interval information.
2. We extend the matching modulo equality algorithm of SMT with a new matching algorithm based on intervals.
3. We present an implementation of our approach in the Alt-Ergo SMT solver. Results on benchmarks coming from deductive verification frameworks show that our strategy competes with tools based on other techniques.

Outline: We illustrate the general ideas of our approach by running a motivating example in Section 2. The design of the first layer is presented in Section 3. Section 4 gives an overview of the axioms about rounding, and intervals matching is detailed in Section 5. Benchmarks are given and discussed in Section 6 and we conclude in the last section.

2 Motivating Example

To illustrate the basic idea of our method, we consider a second simple example. We assume two binary32 FP variables u and v, a rounding operator $\circ(\cdot)$ and a mapping function $\overline{\cdot}$ from FP to reals. We will try to establish an upper bound

for absolute error of the FP addition $u \oplus_m v$, assuming initial bounds for u and v. This (validity) problem can be expressed as follows:

$$(\texttt{2.} \preceq u \preceq \texttt{10.} \ \wedge \ \texttt{2.} \preceq v \preceq \texttt{10.}) \implies (\overline{u \oplus v}) - (\overline{u} + \overline{v}) \leq 0.00000096$$

where `2.` and `10.` are syntactic sugar for the finite FP constants whose real values are 2 and 10 respectively. Note that we omit the rounding mode m when it is the default one or when it is irrelevant.

To establish the validity of the example with our approach, we rely on a three-layer extension of Alt-Ergo as shown in Figure 1.

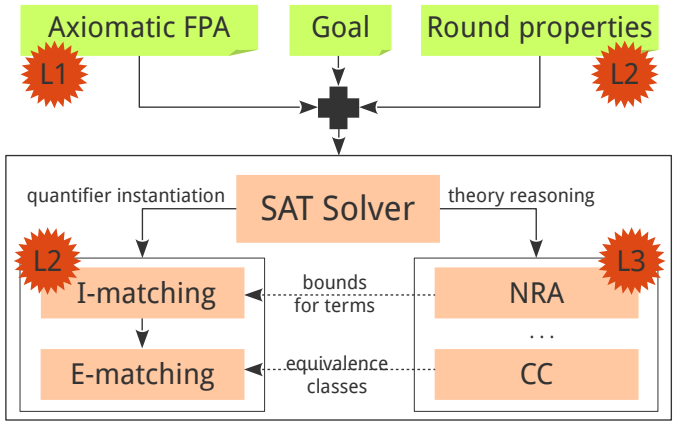

Fig. 1: A view of our framework inside the Alt-Ergo SMT solver. L1, L2, and L3, represent the different layers of our extension.

The first layer L1 consists of a set of "generic" axioms about FPA. It makes it possible to reason about the shape of FP expressions and to reduce FP constraints to real ones, using the rounding operator, when FP numbers are finite. More details about this layer are given in Section 3.

The second layer L2 plays an important role in our approach, as it enables effective cooperation between FP and real reasoning. It is made of two parts: an axiomatic part that provides properties about the rounding operator in an extended syntax of Alt-Ergo (see Section 4), and an interval-matching mechanism to instantiate these axioms (see Section 5).

The third layer L3 is a reasoning engine for non-linear real arithmetic that is able to provide bounds for terms. It is extended to partially handle some operators like rounding, exponentiation, integer logarithm, maximum, and absolute value. More precisely, this consists in enabling calculations when these operators are applied to constants, as we will see in the next sections.

To prove our example, six axioms from layer L1 and one axiom from layer L2 are required. These axioms are shown in Figure 2. L1 axioms rely on a set of uninterpreted predicate symbols (`is_finite`, `is_infinite`, `is_nan`, `is_positive`,

(L1-1) $\forall z.\ \texttt{in\_range}(z) \iff -\texttt{0x1.FFFFFEp127} \leq z \leq \texttt{0x1.FFFFFEp127}$

(L1-2) $\forall m.\forall x.\forall y.\ (\texttt{is\_finite}(x) \wedge \texttt{is\_finite}(y) \wedge \texttt{in\_range}(\circ_m(\overline{x}+\overline{y}))) \implies \overline{x \oplus_m y} = \circ_m(\overline{x}+\overline{y})$

(L1-3) $\forall x.\forall y.\ x \preceq y \implies \bigvee \begin{pmatrix} \texttt{is\_finite}(x) \wedge \texttt{is\_finite}(y) \\ \texttt{is\_infinite}(x) \wedge \texttt{is\_negative}(x) \wedge \neg\texttt{is\_nan}(y) \\ \texttt{is\_infinite}(y) \wedge \texttt{is\_positive}(y) \wedge \neg\texttt{is\_nan}(x) \end{pmatrix}$

(L1-4) $\forall x.\forall y.\ (\texttt{is\_finite}(x) \wedge \texttt{is\_finite}(y) \wedge x \preceq y) \implies \overline{x} \leq \overline{y}$

(L1-5) $\forall x.\ (\texttt{is\_infinite}(x) \vee \texttt{is\_nan}(x)) \implies \neg\ \texttt{is\_finite}(x)$

(L1-6) $\forall x.\ \neg\ (\texttt{is\_negative}(x) \wedge \texttt{is\_positive}(x))$

(L2-1) $\forall z.\forall i.\forall j.\quad i \leq z \leq j \implies -2^{\alpha} \leq \circ(z) - z \leq 2^{\alpha}$
where $\alpha = \texttt{ilog}_2(\max(|i|, |j|, 2^{e_{\min}+prec-1})) - prec$

Fig. 2: Overview of axioms from L1 and L2 needed to prove the example.

and `is_negative`) that are used to characterize the shape of FP expressions. L1-1 defines the range of real numbers that can be represented by finite binary32 FP numbers. When inputs are finite and the result does not overflow, the meaning of addition in terms of rounding operator is given by L1-2. Axioms L1-3 and L1-4 deal with the $\preceq$ relation: the first axiom enumerates the possible cases for a hypothesis $x \preceq y$, and the second one maps this predicate to its real counterpart when the arguments are finite. As for L1-5 and L1-6, they just impose some restrictions on the uninterpreted predicates. Finally, you may notice that axiom L2-1 from layer L2 is much more sophisticated. It bounds the distance between a real and its rounded value (when rounding to nearest) using some parameters *prec* and $e_{\min}$ that depend on the format (see Section 3); function $\texttt{ilog}_2$ denotes the integer part of the base-2 logarithm.

The reasoning steps needed to prove the initial example within our framework are shown in Figure 3. The initial hypotheses are named as follows: H1 $\equiv$ `is_finite(2.)`, H2 $\equiv$ `is_finite(10.)`, H3 $\equiv$ `2.` $\preceq u$, H4 $\equiv$ `2.` $\preceq v$, H5 $\equiv u \preceq$ `10.`, and H6 $\equiv v \preceq$ `10.`. A property deduced at step i is named Di. Steps from 1 to 10 prove that u and v are finite using mainly E-matching (EM) and Boolean reasoning. Steps from 11 to 14 deduce bounds for $\overline{u}$ and $\overline{v}$ from those of u and v using the same reasoning engines. Step 15 instantiates the axiom that reduces FP addition to real addition, plus a rounding operation. However, to be able to use its conclusion, we should first show that the addition does not overflow: given bounds for $\overline{u}+\overline{v}$ (step 16), intervals-matching generates an instance from L2-1, which is simplified by the SAT (step 17). Thanks to an instance of L1-1, a combination of arithmetic and Boolean reasoning makes it possible to

IDs	used hypotheses	used axioms	used reasoners	produced deductions
1	H1	L1-5	EM, SAT	$\neg\texttt{is\_infinite}(2.)$
2	H1	L1-5	EM, SAT	$\neg\texttt{is\_nan}(2.)$
3	H2	L1-5	EM, SAT	$\neg\texttt{is\_infinite}(10.)$
4	H2	L1-5	EM, SAT	$\neg\texttt{is\_nan}(10.)$
5	H3, D{1,2}	L1-3	EM, SAT	$\texttt{is\_finite}(u) \vee (\texttt{is\_infinite}(u) \wedge \texttt{is\_positive}(u))$
6	H4, D{1,2}	L1-3	EM, SAT	$\texttt{is\_finite}(v) \vee (\texttt{is\_infinite}(v) \wedge \texttt{is\_positive}(v))$
7	H5, D{3,4}	L1-3	EM, SAT	$\texttt{is\_finite}(u) \vee (\texttt{is\_infinite}(u) \wedge \texttt{is\_negative}(u))$
8	H6, D{3,4}	L1-3	EM, SAT	$\texttt{is\_finite}(v) \vee (\texttt{is\_infinite}(v) \wedge \texttt{is\_negative}(v))$
9	D{5,7}	L1-6	EM, SAT	$\texttt{is\_finite}(u)$
10	D{6,8}	L1-6	EM, SAT	$\texttt{is\_finite}(v)$
11	H{1,3}, D9	L1-4	EM, SAT	$2 \leq \overline{u}$
12	H{1,4}, D10	L1-4	EM, SAT	$2 \leq \overline{v}$
13	H{2,5}, D9	L1-4	EM, SAT	$\overline{u} \leq 10$
14	H{2,6}, D10	L1-4	EM, SAT	$\overline{v} \leq 10$
15	D{9,10}	L1-2	EM, SAT	$\texttt{in\_range}(\circ(\overline{u}+\overline{v})) \Rightarrow \overline{u \oplus v} = \circ(\overline{u}+\overline{v})$
16	D{11,12,13,14}		NRA	$\overline{u}+\overline{v} \in [4;20]$
17	D16	L2-1	EM, IM, SAT	$-2^{-20} \leq \circ(\overline{u}+\overline{v}) - (\overline{u}+\overline{v}) \leq 2^{-20}$
18	D{16,17}		NRA	$4 - 2^{-20} \leq \circ(\overline{u}+\overline{v}) \leq 20 + 2^{-20}$
19	D{18}	L1-1	EM, SAT, NRA	$\texttt{in\_range}(\circ(\overline{u}+\overline{v}))$
20	D{15,19}		SAT	$\overline{u \oplus v} = \circ(\overline{u}+\overline{v})$
21	D{17,20}		NRA	$\overline{u \oplus v} - (\overline{u}+\overline{v}) \leq 2^{-20}$

Fig. 3: Reasoning steps used to prove the example.

deduce that the addition does indeed not overflow (steps 18 and 19). Finally, using the deductions of steps 17 and 20, we derive that $\overline{u \oplus v} - (\overline{u}+\overline{v}) \leq 2^{-20}$, which concludes the proof, since $2^{-20} = 0.000000953\cdots < 0.00000096$.

3 Layer 1: Generic Axiomatic FPA

The IEEE-754 standard defines several formats for representing FP numbers and gives the semantics of arithmetic operators over data of these formats [1]. The formats are specified at various abstraction levels. At Level 1, formats are just some subsets of real numbers to which $-\infty$ and $+\infty$ are added. At Level 2, zero is split into a positive zero and a negative zero, and a set of *Not-a-Number* (NaN) is added. The signed zeros behave as the multiplicative inverse of the infinities, while NaNs are the result of invalid operations such as $0 \times \infty$ or $\infty - \infty$. At Level 3, finite values get represented by their sign, significant, and exponent. The set of NaNs is also further refined. At Level 4, FP data get mapped to strings of bits. The IEEE-754 standard supports both binary and decimal formats, but we will focus on binary formats only.

The SMT-LIB theory follows a similar approach to describing FP formats [5], though it does not provide all the features of the IEEE-754 standard. Indeed, the latter is underspecified when it comes to NaNs: the standard tells when the result of an operation is NaN, but it does not tell which *payload* the NaN carries. So the SMT-LIB theory chose to have only one NaN to account for all the possible implementations. This also means that the theory provides a function for going from strings of bits to FP data, but not the other way around.

Taking advice of both the IEEE-754 standard and the SMT-LIB theory, we have formalized FP formats as a Why3 theory. The first step is to define an abstract type `t` which will be later instantiated for each format. We also give a few abstract predicates on this type:

```
predicate is_finite    t
predicate is_infinite  t
predicate is_nan       t
```

The Why3 syntax here means that these predicates take a single argument of type `t`. Then there are two axioms to make sure that a FP datum is either finite or infinite or NaN, of which axiom L1-5 of Figure 2 is an implication.

```
axiom is_not_nan: forall x:t.
  (is_finite x \/ is_infinite x) <-> not (is_nan x)
axiom is_not_finite: forall x:t.
  not (is_finite x) <-> (is_infinite x \/ is_nan x)
```

Note that our formalization uses abstract predicates, but for SMT solvers that support enumerations, one could also imagine a less abstract formalization that does not require these two axioms.

Any finite FP datum has an associated real value. These values are of the form $m \cdot \beta^e$ with m an integer significand, e an exponent, and β the radix. The radix is 2 for the formats we are interested in. The ranges of allowed m and e are given by the format in the following way:

$$|m| < \beta^{prec} \wedge e_{\min} \leq e \wedge |m \cdot \beta^e| < \beta^{e_{\max}}.$$

The values of *prec*, $e_{\min}$, and $e_{\max}$ can be recovered from the Level 4 description of a format. For instance, if a binary format uses e_w bits for representing the exponent and m_w bits for representing the mantissa (that is, the significant without its most-significant bit for normal numbers), then

$$prec = m_w + 1 \wedge e_{\max} = 2^{e_w - 1} \wedge e_{\min} = 3 - e_{\max} - prec.$$

Our formalization does not give direct access to the significand or the exponent. It just provides an abstract function from `t` to the set of a real numbers, which is the function that was denoted $\overline{\cdot}$ in the previous section.

```
function to_real t : real
```

As for formats, the formalization uses an abstract real for the largest representable number (that is, $\beta^{e_{\max}} \cdot (1 - \beta^{-prec})$) and defines a predicate for characterizing real numbers in the range of finite numbers.

```
constant max_real : real
predicate in_range (x:real) = -max_real <= x <= max_real
```

Not all the real numbers that are in range are representable, so we arrive to the notion of rounding. It is present in both the IEEE-754 standard and the SMT-LIB theory. Let us first enumerate the five rounding modes supported by the standard:

```
type mode = RNE | RNA | RTP | RTN | RTZ
```

The first two modes round to nearest, with tie breaking to even (RNE) or away from zero (RNA). The last three modes are directed rounding: toward $+\infty$ (RTP), toward $-\infty$ (RTN), and toward zero (RTZ).

For a given target format, a rounding operator tells us which FP number should be used to represent a real number. If the real number is representable, then the choice is trivial; the only peculiarities come from the existence of signed zeros, which we will not detail here. Otherwise the FP number should be the closest one according to the rounding mode. For now, we follow the examples of the Gappa tool [8] and of the Flocq library [3], so we assume that the target format has an unbounded $e_{\max}$. That way, we do not have to bother with the issue of overflow yet. So the rounding operator can be given the following signature.

```
function round mode real : real
```

This is yet again declared as an abstract function, though it has a mathematical definition. If we denote $\circ_m$ the rounding operator with mode m, then we have

$$\circ_m(x) = \lceil x/\beta^e \rfloor_m \cdot \beta^e \tag{2}$$

with $e = \max(e_{\min}, \lfloor \log_\beta |x| \rfloor + 1 - prec)$. The integer part $\lceil \cdot \rfloor_m$ selects the integer the closest to a given real according to mode m. This definition of the rounding operator is sufficient to prove all the properties of Section 4, *e.g.* monotonicity of the rounding operator, as done in Flocq.

Note that, if one wanted to use such a concrete definition in an SMT solver, one would also need $\log_\beta$ to be defined. This can be avoided by instead providing a function from $|x|$ to β^e, since this function can be finitely axiomatized using just inequalities for any format, as was done in [14]. In our case, we do not care about these details, as `round` is kept as an abstract function.

Our rounding operator ignores the issue of overflow, but this issue still has to be handled. We do so by defining the following predicate. It will be used to decide whether a rounded value is relevant or not as a result of an FP arithmetic operator.

```
predicate no_overflow (m:mode) (x:real) =
  in_range (round m x)
```

We now have enough definitions to state what the behavior of the FP addition is when its inputs are finite numbers. More precisely, the IEEE-754 standard states that an addition "shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded

that result". This trivially translates into the following specification (axiom L1-2 of Figure 2) where + denotes the addition over real numbers.

```
function add mode t t : t
axiom add_finite: forall m:mode, x y:t.
  is_finite x -> is_finite y ->
  no_overflow m (to_real x + to_real y) ->
  is_finite (add m x y) /\
  to_real (add m x y) = round m (to_real x + to_real y)
```

As for the overflow case, here is a small excerpt of the `add_special` axiom which covers all the exceptional behaviors of the FP addition according to the standard:

```
axiom add_special: forall m:mode, x y:t.
  let r = add m x y in
  ... (* 6 other conjuncts *) ... /\
  (is_finite x /\ is_finite y /\
   not no_overflow m (to_real x + to_real y)
   -> same_sign_real r (to_real x + to_real y) /\
      overflow_value m r)
```

The `same_sign_real` relation tells which sign the final result has, assuming the infinitely-precise intermediate result is nonzero (which is the case if it overflowed). The `overflow_value` relation then selects the final result depending on the rounding mode and its sign.

An important property of our formalization, which we hope is apparent from the excerpts, is that it is a straightforward translation from the IEEE-754 standard. That does not mean that it is trivially error-free though, since we could well have forgotten some hypotheses or we could have mistyped an `r` for an `x` for instance. To prevent this issue, we have realized this formalization using the Coq proof assistant. What this means is that we have given concrete definition to all the abstract definitions (*e.g.* the `Bplus` function of Flocq for the FP addition) and then Why3 required us to formally prove in Coq that all the axioms hold. This gives a high level of confidence in the correctness of our formalization. A more detailed presentation of Layer 1 is given in this technical report [11].

4 Layers 2 and 3: Rounding Properties

Given a constraint mixing the rounding operator with non-linear real arithmetic terms, one can reduce it to an extension of non-linear real arithmetic with floor and ceiling functions using the encoding of the rounding operator given by Equation (2). Then a solver that supports mixed real/integer non-linear arithmetic can be used to solve the resulting problem, as was done in [14].

In this paper, we adopt a different approach: instead of fully encoding the rounding operator, we reason modulo a set of its properties. Although theoretically incomplete compared to mixed non-linear integer/real encoding, our

(L2-1) $\forall x.\forall i.\forall j. \quad i \leq x \leq j \Rightarrow -2^{\alpha} \leq \circ(x) - x \leq 2^{\alpha}$
where $\alpha = \mathtt{ilog_2}(\max(|i|, |j|, 2^{e_{\min}+prec-1})) - prec$

(L2-2) $\forall x.\forall i.\forall j.\forall m. \;\; i \leq x \leq j \Rightarrow \circ_m(i) \leq \circ_m(x) \leq \circ_m(j)$

(L2-3) $\forall x.\forall i.\forall j.\forall m. \;\; i \leq x \leq j \Rightarrow \circ^{\mathbb{Z}}_m(i) \leq \circ^{\mathbb{Z}}_m(x) \leq \circ^{\mathbb{Z}}_m(j)$

(L2-4) $\forall x.\forall m_1.\forall m_2. \;\; \circ_{m_1}(\circ_{m_2}(x)) = \circ_{m_2}(x)$

(L2-5) $\forall x.\forall y.\forall m. \;\; x \leq y \Rightarrow \circ_m(x) \leq \circ_m(y)$

(L2-6) $\forall x.\forall y.\forall m. \;\; \circ_m(x) < \circ_m(y) \Rightarrow x < \circ_m(y)$

(L2-7) $\forall x.\forall y.\forall m. \;\; \circ_m(x) < \circ_m(y) \Rightarrow \circ_m(x) < y$

(L2-8) $\forall x.\forall i.\forall m. \;\; \circ_m(x) < i \Rightarrow x < \circ_{\mathrm{RTP}}(i)$

(L2-9) $\forall x.\forall i.\forall m. \;\; i \leq \circ_m(x) \Rightarrow \circ_{\mathrm{RTP}}(i) \leq \circ_m(x)$

(L2-10) $\forall x.\forall y.\forall m. \;\; |x| \geq 1 \Rightarrow |x| = 2^{\mathtt{ilog_2}(|x|)} \Rightarrow \circ_m(x \cdot \circ_m(y)) = x \cdot \circ_m(y)$

(L2-11) $\forall x.\forall y.\forall m. \;\; \circ_m(x) < \circ_m(y) \Rightarrow \circ_m(\circ_m(x) - \circ_m(y)) < 0$

(L2-12) $\forall x.\forall y.\forall m. \;\; \circ_m(x) < -\circ_m(y) \Rightarrow \circ_m(\circ_m(x) + \circ_m(y)) < 0$

Fig. 4: Overview of the main axioms of Layer 2.

experimental evaluation shows that the method is effective in practice, in particular in the domain of program verification.

An overview of the rounding properties that are provided by Layer 2 are given in Figure 4. Most of these axioms are borrowed from the Flocq formalization of FPA for the Coq proof assistant. Axiom L2-1 bounds the distance between a real and its rounded value (*i.e.* nearest float) with respect to the default rounding mode. Similar properties for other rounding modes are also provided. Axiom L2-2 states the monotonicity of rounding. This particular formulation is used to deduce numerical bounds for rounded values as we will see in the next section. Axiom L2-3 is similar to L2-2 except that operator $\circ^{\mathbb{Z}}$ rounds to the nearest integer. Axiom L2-4 states the idempotency of rounding, independently of the rounding mode. Classical formulation of monotonicity is given by axiom L2-5, and the three next axioms are consequences of L2-5's contrapositive (modulo idempotency). Axiom L2-9 makes it possible to improve lower bounds of rounded expressions. A similar axiom is used to tighten upper bounds. The last three axioms provide some situations where arithmetic expressions can be simplified.

Several of those axioms are meant to be instantiated using some constant values (*e.g.* any occurrence of i and j) so that numerical computations can be performed. For instance, axiom L2-2 is akin to a *propagator* as found in CP techniques, while axiom L2-8 acts as a *contractor*. To do so, the SMT solver has to be able to compute an expression such as $\circ_m(i)$ when i is a constant. We thus put a shallow preprocessor in front of the NRA engine to reduce these

expressions to rational numbers that the NRA engine can make use of. This involves providing some code for evaluating $\circ$, $\circ^{\mathbb{Z}}$, $2^{(\cdot)}$, $\texttt{ilog}_2$, max, $|\cdot|$.

5 Interval-Based Instantiation

Most of the axioms given in Figure 4 cannot be efficiently instantiated using generic E-matching techniques of SMT solvers, as it is not always possible to provide good triggers for them. A trigger for an axiom $\forall \boldsymbol{x}.\varphi(\boldsymbol{x})$ is a term (or a set of terms) that covers all variables $\boldsymbol{x}$. Based on this definition, a trigger-inference algorithm would presumably compute $\{\circ_m(x), \circ_m(i), \circ_m(j)\}$ for the L2-2 axiom:

$$\text{(L2-2)} \quad \forall x.\forall i.\forall j.\forall m.\ \ i \leq x \leq j \Rightarrow \circ_m(i) \leq \circ_m(x) \leq \circ_m(j).$$

This set of terms, however, is not suitable for this axiom, as it would prevent its application when proving the following formula:

$$2 \leq a \leq 4 \implies 2 \leq \circ(a) \leq 4.$$

Indeed, the only way to instantiate the trigger is by choosing $x = i = j = a$ since the only rounded value in the formula is $\circ(a)$. Yet the proper way to instantiate the axiom is $x = a$, $i = 2$, $j = 4$. Then effectively computing $\circ(2)$ and $\circ(4)$ makes it possible to conclude.

In order to efficiently reason modulo our rounding properties, we have decorated them with two kinds of triggers: *syntactic* and *interval* triggers. Syntactic triggers are those that are already used by the generic E-matching mechanism. For instance, the set of syntactic triggers of axiom L2-2 is the singleton $\{\circ_m(x)\}$. Interval triggers are guards used to carefully instantiate the variables that are not covered by syntactic triggers. The purpose of this kind of triggers is twofold: (a) get rid of permissive or restrictive syntactic triggers, (b) take current arithmetic environment into account to guide the instantiation process. For instance, a suitable interval trigger for axiom L2-2 is the set $\{i \leq x, x \leq j\}$.

The language of interval triggers is defined by the following grammar:

$$its ::= it \mid it, its \qquad it ::= b\ \mathcal{R}\ t \mid t\ \mathcal{R}\ b \qquad b ::= c \mid i \qquad \mathcal{R} ::= \leq \mid <$$

where c is a numerical constant, i is a quantified variable, and t is a term. In the rest of the paper, we assume that all the variables of the t terms (*resp.* none of the variables in the b bounds) appear in syntactic triggers.

Let us illustrate the use of triggers with the Alt-Ergo statement of axiom L2-8 from Figure 4. (It has been simplified a bit for the sake of readability.)

```
axiom monotonicity_contrapositive_1 :
  forall x, i : real. forall m : mode
  [ round(m, x) ] { round(m, x) < i }.
  x < round(RTP, i)
```

The term between square brackets is a syntactic trigger: variables m and x will be instantiated using any ground term $\circ(\cdot)$ found in the context. The term between curly brackets is both an interval trigger and a hypothesis of the axiom: variable i will be instantiated by any constant bound such that $\circ_m(x) < i$ holds.

Note that interval triggers are instantiated by querying the NRA engine, once syntactic triggers have been instantiated. For instance, let us suppose that the syntactic trigger gives $\circ_m(x)$ and that the NRA engine then tells that $\circ_m(x) < 5$ holds. Thus any number that is larger or equal to 5 satisfies the interval trigger $\circ_m(x) < i$. Since there are no other interval triggers about i, the most precise instance of i is 5. This leads to the following instance of the axiom being added:

$$\circ_m(x) < 5 \implies x < 5$$

since $\circ_{\mathrm{RTP}}(5)$ evaluates to 5.

Now let us suppose that the current upper bound on $\circ_m(x)$ is just $\circ_m(x) \leq 5$. This time, any number that is strictly larger to 5 satisfies the interval trigger. There is no longer any most precise instance. So Alt-Ergo selects some arbitrary number $\varepsilon > 0$ and instantiates i with $5 + \varepsilon$. So some incompleteness creeps here. However, if ε is small enough, then $\circ_{\mathrm{RTP}}(5 + \varepsilon)$ evaluates to the successor of 5 in the FP format, which leads to adding the following axiom instance:

$$\circ_m(x) < 5 + \varepsilon \implies x < 5 + 2^{3-prec}.$$

To conclude the section, some axioms of Layer 2 with their interval triggers are shown in Figure 5. You may note that: (a) the verbose expression `let k = ...` in the second axiom actually reduces to a numerical constant by calculation when instantiating `i` and `j` with numerical values, (b) in the interval triggers of the third axiom, the bounds are constants. In this case, interval matching consists in simple numerical checks.

6 Implementation and Evaluation

Implementation. To evaluate our approach, we have extended Alt-Ergo's instantiation engine with an interval-matching mechanism. The arithmetic engine has been strengthened to enable calculation with some function symbols (rounding operator, exponentiation, integer logarithm, etc) when applied to numerical constants. Rounding properties are given as a list of axioms in a separate file, annotated with their triggers. Generic FPA axioms are provided by a Why3 FPA theory. Currently, the instances generated from the axioms of Layer 2 are given back to the SAT solver. The implementation will be available in the next release of Alt-Ergo.

Tools and Benchmarks Description. In addition to our implementation (denoted AE and AE+G in the result tables), we use three solvers for the evaluation: Z3 (v. 4.5.0), MathSAT5 (v. 5.3.14, denoted MS+A and MS+B), and Gappa

```
 axiom integer_rounding_operator :
   forall x, i, j : real. forall m : mode.
   [ int_round(m, x) ] { i <= x, x <= j }.
   int_round(m, i) <= int_round(m, x) <= int_round(m, j)

axiom rounding_operator_absolute_error_RNE :
   forall x, i, j : real.
   [ round(RNE, x) ] { i <= x, x <= j }.
   let k =
    pow(2.,
     ilog2(max(abs(i),abs(j),pow(2., emin+prec-1))) - prec)
    in -k <= round(RNE, x) - x <= k

 axiom round_of_int:
   forall x : int. forall m : mode.
   [ round(m, real_of_int(x))]
   { 0. <= real_of_int(x) + pow(2., prec),
     real_of_int(x) - pow(2., prec) <= 0. }.
   round(m, real_of_int(x)) = real_of_int(x)
```

Fig. 5: Some axioms of Layer 2 in Alt-Ergo's extended syntax.

(v. 1.3.1). While not an SMT solver, Gappa is included because it has significantly inspired our approach. We do not include REALIZER [14] as its input language is too limited to talk about infinities and NaNs.

Considered benchmarks consist of two sets of verification conditions (VC) extracted from numerical C[5] and SPARK programs, and of the QF-FP category of SMT-LIB benchmarks. Figure 6 explains how VCs are encoded to different solvers. In particular, notice that SMT-LIB benchmarks are directly fed to Z3 and MathSAT5, but they are translated to Why3 and combined with an axiomatic FPA before they are given to Alt-Ergo and Gappa.

VCs from C and SPARK programs are first extracted using Why3's weakest precondition calculus. They are then combined with generic FPA axioms and encoded to different solvers. For Z3 and MathSAT5, a faithful mapping from Why3 FPA axiomatization to the FP theory of SMT-LIB is used. In this case, first-order axiomatization of FP operations is removed. More generally, since MathSAT5 does not handle quantifiers, Why3 always eliminates the quantified parts of the VCs sent to it. The same elimination process is used for Gappa, except that Layer-1 axioms are first instantiated by Why3 since Gappa only knows about real numbers. This means that a set of ground instances is generated from FP operations' axioms using matching techniques and initial ground terms

[5] See `https://www.lri.fr/~sboldo/research.html` and `http://toccata.lri.fr/gallery/fp.en.html` for more details.

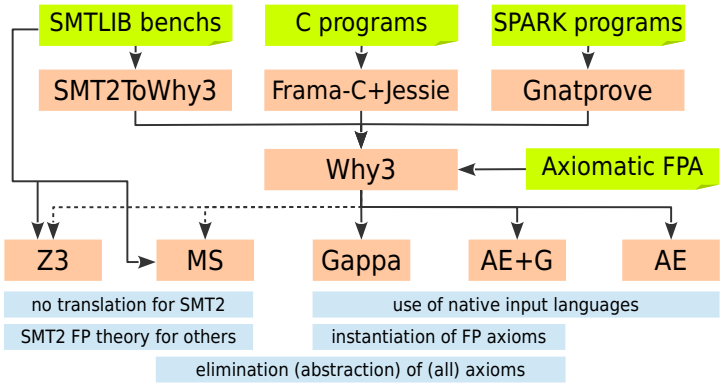

Fig. 6: Generation of verification conditions in the language of different solvers.

of the VCs.[6] To better evaluate the cost of Layer-1 axioms and to provide a fairer comparison with MathSAT5 and Gappa, we also generate Alt-Ergo benchmarks (denoted AE+G, where G stands for *ground*) where FP operations' axioms are instantiated and eliminated the same way they are for Gappa.

Since not all the VCs extracted for C and SPARK programs need FPA reasoning for them to be proved, we have filtered them as follows: if a VC is proved by Alt-Ergo without any reasoning modulo generic FPA axioms, rounding properties, or any other extension, it is removed from the benchmark. Filtering yields test-suites of 307 VCs (out of 1848 initially) from C programs and 1980 VCs (out of 23903) from SPARK programs. It should be noted that: (a) some VCs extracted from C and SPARK programs may require a combination of FPA reasoning with other theories and quantifiers instantiation to be proved, and (b) all the VCs are not provable.

For QF-FP benchmarks, we only keep formulas that are not known to be *satisfiable*, as we are interested in unsatisfiability (or dually in validity) in the context of program verification. Consequently *Wintersteiger* benchmarks with a SAT status, and *Griggio* benchmarks that are shown to be satisfiable by Z3 or MathSAT5 are filtered out. Schanda benchmarks are also discarded as they are already included in SPARK programs.

Experimental Results. The results of our evaluation are shown in Figures 7 and 8. Time and memory limits were set to 60 seconds and 3GB, respectively. For each prover (in the columns), we report the number and the percentage of proved VCs, and the time required for the proofs. Clusters CMP-1 and CMP-3 contain problems generated by Why3. Clusters CMP-2 and CMP-4 are also generated by Why3 but the resulting VCs are quantifier-free; in the case of AE+G and Gappa, Layer-1 axioms are automatically instantiated by Why3 before being eliminated.

[6] Having Why3 instantiate some axioms in the VCs sent to Gappa is not specific to these benchmarks.

Cluster CMP-5 contains the original problems from SMT-LIB competition without going through Why3. We report in row "loc. unique" the number of formulas that a given solver exclusively discharges compared to other solvers in the same CMP-i cluster. Similarly, we report in "glob. unique" the number of exclusive proofs of each solver. Finally, we report the number of VCs that are proved by at least one prover in line "total".

		CMP-1		CMP-2			
		AE	Z3	AE+G	Gappa	MS+B	MS+A
C	proved	194	2	194	199	4	2
	% proved	63.19%	0.65%	63.19%	64.82%	1.30%	0.65%
	time (secs)	566	< 1	348	78	4	< 1
	loc. unique	192	0	21	28	0	0
	glob. unique	8	0	2	25	0	0
	total	230/307 (74.92%) proved with at least one solver					
SPARK	proved	806	720	541	488	170	13
	% proved	40.71%	36.36%	27.32%	24.65%	8.59%	0.66%
	time (secs)	3090	4142	4769	305	301	1
	loc. unique	244	157	66	33	100	2
	glob. unique	151	105	31	33	7	0
	total	1136/1980 (57.37%) proved with at least one solver					

Fig. 7: Results on filtered VCs extracted from C and SPARK programs.

		CMP-3	CMP-4		CMP-5		
		AE	AE+G	Gappa	Z3	MS+B	MS+A
Wintersteiger unsat	proved	20012	19863	18102	20035	17201	17200
	% proved	99.89%	99.14%	90.35 %	100%	85.85%	85.85%
	time (secs)	876	487	44	65	66	63
	loc. unique	-	1784	23	2834	0	0
	glob. unique	0	0	0	0	0	0
	total	20035/20035 (100%) proved with at least one solver					
Griggio unknown + unsat	proved	2	33	0	50	49	5
	% proved	1.75%	28.95%	0%	43.86%	42.98%	4.39%
	time (secs)	18	621	0	1337	723	1
	loc. unique	-	33	0	6	3	1
	glob. unique	0	1	0	0	2	0
	total	57/114 (50.00%) proved with at least one solver					

Fig. 8: Results on QF-FP category of SMT-LIB benchmarks.

We notice in Figure 7 that Alt-Ergo and Gappa outperform other solvers on the C benchmark. Indeed, the specification of the corresponding C programs heavily uses reals for expressing pre- and post-conditions about FP numbers, which impedes solvers that do not handle well the combination of FP and real

reasoning. In particular, the `fp.to_real` function symbol is hardly supported. While Z3 parses it properly, it has troubles in solving problems involving it.[7]

On the opposite, SPARK programs are specified using FP numbers. Consequently, Z3's results are much closer to those of Alt-Ergo. Surprisingly, MathSAT5 with either bit-blasting (MS+B) or ACDCL (MS+A) does not perform well. This is probably due to the fact that MathSAT5 is not well tuned for the context of deductive program verification. One can notice two other interesting results on the SPARK benchmarks: first, AE+G's success rate is close to Gappa's but quite far from Alt-Ergo's. This is certainly due to the abstraction of quantified formulas, which prevents AE+G (and Gappa) from dynamically generating relevant instances to discharge some proofs. Second, the solvers are highly complementary. For instance, the SPARK results of Figure 7 show that Alt-Ergo and Z3 together solve 963 VCs, but more than a third ($244 + 157$) of these VCs are solved by only one prover.

Wintersteiger unsat benchmarks are very simple crafted formulas meant to stress compliance to the FP SMT-LIB standard and detect bugs in implementations. Z3 proves all the formulas of the benchmark. For Alt-Ergo, 23 formulas are not proved due to the inaccuracy of its NRA engine when bounding terms involving square root. For MathSAT5, unproved formulas are related to `fma` and `remainder` operators, which are apparently not supported yet. Gappa fails to prove approximately 2000 formulas due to the default reduced precision of its internal domains, since the infinitely-precise result of a binary64 `fma` might require up to 3122 bits so that its rounded value can be computed exactly.

Bit-blasting based techniques perform better on Griggio benchmarks. Since AE+G outperforms AE, the bad results likely come from a poor interaction between the E-matching and SAT engines of Alt-Ergo. As for Gappa, its inability to handle these benchmarks is due to the absence of any kind of SAT solver.

7 Conclusion and Future Work

We have presented a three-layer axiomatic strategy for reasoning about floating-point arithmetic in SMT. Experimental evaluations show that it is effective in practice for reasoning in a combination of FPA and real numbers, as required for instance to refer to the distance between FP and real computations.

We opted for the presented solution for different reasons: (a) it is lightweight, less intrusive and requires minimal changes in the solver, (b) potential bugs in most of the added code (mainly in interval matching) do not impact the soundness of our results, (c) axioms of Layer 1 have been verified with the Coq proof assistant for extra confidence, (d) rounding properties in Layer 2 are human-readable, can be reviewed and validated by experts, and are easily extensible, (e) our solution can be trivially extended to support decimal FP formats if need arises.

Future work. An interesting direction of research for improving performances could be to implement the generic FPA part as a built-in procedure, and inlining

[7] `https://github.com/Z3Prover/z3/issues/14`

most of rounding properties as CP-like propagators. As the benchmarks show, the various approaches are complementary. We are thus planning to see how bit-blasting, ACDCL, and RIA techniques could be integrated in our framework, in particular to handle satisfiable formulas.

References

1. IEEE Standard for Floating-Point Arithmetic. Technical report, IEEE, Aug. 2008.
2. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.
3. S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In E. Antelo, D. Hough, and P. Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011.
4. M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, pages 1–33, 2013.
5. M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In J.-M. Muller, A. Tisserand, and J. Villalba, editors, *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic*, pages 160–167. IEEE, 2015.
6. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In N. Piterman and S. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013.
7. M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 169–184, Edinburgh, Scotland, UK, 2001.
8. F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *Transactions on Computers*, 60(2):242–253, 2011.
9. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, 2008.
10. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.
11. C. Fumex, C. Marché, and Y. Moy. Automated verification of floating-point computations in Ada programs. Research Report RR-9060, Inria Saclay–Île-de-France, Apr. 2017.
12. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *Proceedings of the 11th European Symposium on Programming (ESOP)*, pages 209–212, Grenoble, France, 2002.
13. J. Harrison. A machine-checked theory of floating point arithmetic. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 113–130, Nice, France, 1999.

14. M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl. Make it real: Effective floating-point reasoning via exact arithmetic. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4, Dresden, Germany, Mar. 2014.
15. C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 524–538, Paphos, Cyprus, 2001.
16. D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3):1–41, May 2008.
17. J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.